MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 587

August 1980

Destructive Reordering of CDR-Coded Lists

by

Guy L. Steele Jr. *

Abstract:

Linked list structures can be compactly represented by encoding the CDR ("next") pointer in a two-bit field and linearizing list structures as much as possible. This "CDR-coding" technique can save up to 50% on storage for linked lists.

The RPLACD (alter CDR pointer) operation can be accommodated under such a scheme by using indirect pointers. Standard destructive reordering algorithms, such as REVERSE and SORT, use RPLACD quite heavily. If these algorithms are used on CDR-coded lists, the result is a proliferation of indirect pointers.

We present here algorithms for destructive reversal and sorting of CDR-coded lists which avoid creation of indirect pointers. The essential idea is to note that a general list can be viewed as a linked list of array-like "chunks". The algorithm applied to such "chunky lists" is a fusion of separate array- and list-specific algorithms; intuitively, the array-specific algorithm is applied to each chunk, and the list algorithm to the list with each chunk considered as a single element.

Keywords: list structure, linked lists, compact lists, CDR-coding, LISP, data structures, data representations, sorting, merge sorting, reversing, destructive list operations

CR Categories: 4.34, 4.49, 5.31

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

* Fannie and John Hertz Fellow

Introduction

With the advent of microprogrammable processors (such as the LISP Machine [Greenblatt] [LISP Machine Group]) it has been possible to use highly encoded, bit-efficient representations of data structures without loss of speed. Here we shall be concerned with LISP-style linked lists which are "CDR-coded".

Each LISP list cell conceptually has two pointers, a CAR and a CDR, which in principle can each point to any LISP object. In "traditional" LISP implementations, such as LISP 1.5 [McCarthy], MacLISP [Moon], and InterLISP [Teitelman], such a cell is represented as two full memory addresses; this is a convenient representation for use on general-purpose computers.

In practice, list cells are usually used in highly stylized ways. For example, the CDR is used almost exclusively to point to another list cell, in order to form linked chains of CAR pointers. Most CDR pointers which do not point to list cells point to the atom NIL, which by convention terminates such a linked chain (see Figure 1).

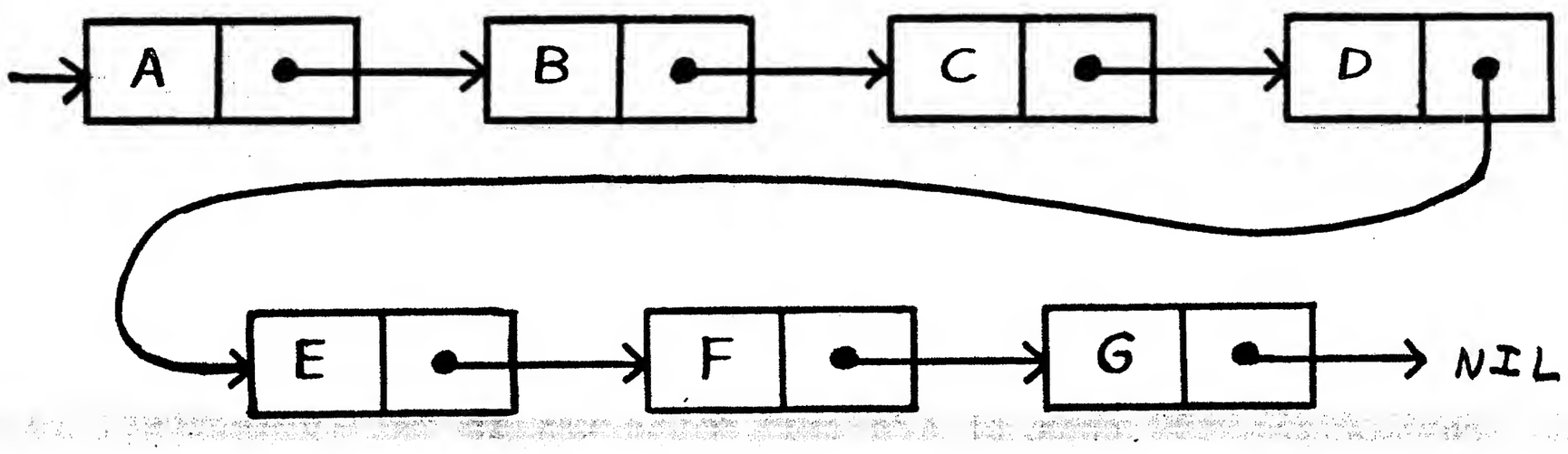

Figure 1
A Standard List of Seven Elements (A B C D E F G)

In fact, studies have shown that a CDR pointer very often points to the sequentially following cell in memory, because of the usual memory allocation strategy, and special "linearizing" allocation strategies typically cause 98% of all CDR pointers to other lists to point to the next cell [Clark]. One can take advantage of this asymmetry of use by encoding the CAR and CDR pointers in different ways. Let memory be divided into cells each of which contains a full memory address for the CAR, but only a two-bit field for the CDR [Hansen] [Greenblatt] [LISP Machine Group]. This two-bit field is used to encode the location of the CDR as follows:

00	(Normal)	The CDR is <u>in</u> the next cell.
01	(Next)	The CDR <u>is</u> the next cell.
10	(NIL)	The CDR is NIL.
11	(Indirect)	The cell has been moved to another location.

If the CDR code in a cell is 00, then the cell is represented in the traditional way; two consecutive cells hold two full addresses, one for the CAR and one for the CDR. (The CDR code in the second cell doesn't matter.) If the CDR code is 01, then the CDR is the cell following the one containing the 01 code. The CDR code 10 means that the CDR is NIL. Using codes 01 and 10, a linked chain of n CAR pointers can be represented as a linear block of n contiguous cells in memory (see Figure 2).

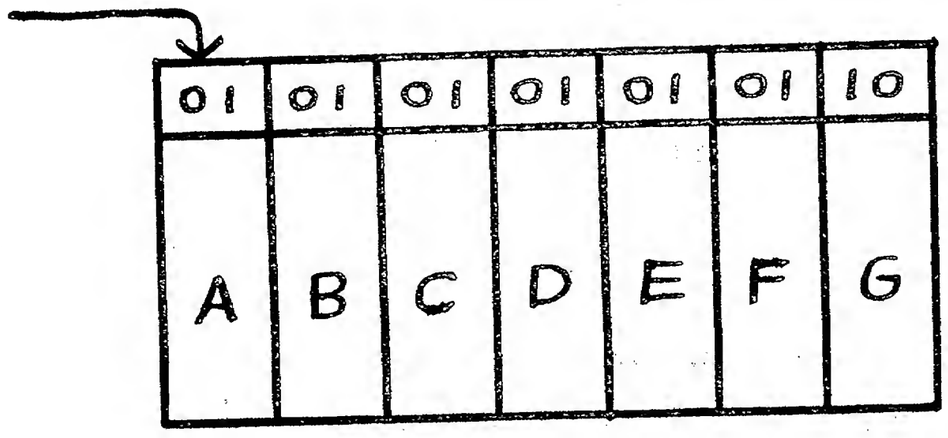

Figure 2
A Compact CDR-Coded List of Seven Elements

We call such a linear block a "CDR-coded list". Assuming the two-bit CDR field to be insignificant compared to the size of a full address, this representation provides a 50% memory saving.

Code 11 is provided for handling the RPLACD operation, that is, for altering the CDR pointer of a list cell. Suppose, in the previous diagram, we wished to alter the CDR pointer of the cell whose CAR is "A" to point to some cell other than the following one (whose CAR is "B"), say to the cell whose CAR is "C". (The effect of doing this would be to splice the second element out of the list, producing the list (A C D E F G).) We cannot simply change the CDR code, because we need to use a Normal CDR code, and the cell whose CAR is "B" may still be in use (pointed to from elsewhere). The solution is to change the CDR code of the cell whose CAR is "A" to Indirect (11), and allocate a pair of consecutive cells elsewhere. The CAR pointer of the cell "A" is altered to point to the new pair, the old CAR pointer (to "A") is copied into the CAR of the new pair, and the CDR pointer of the new pair is made to point to the cell whose CAR is "C". The new situation is shown in Figure 3.

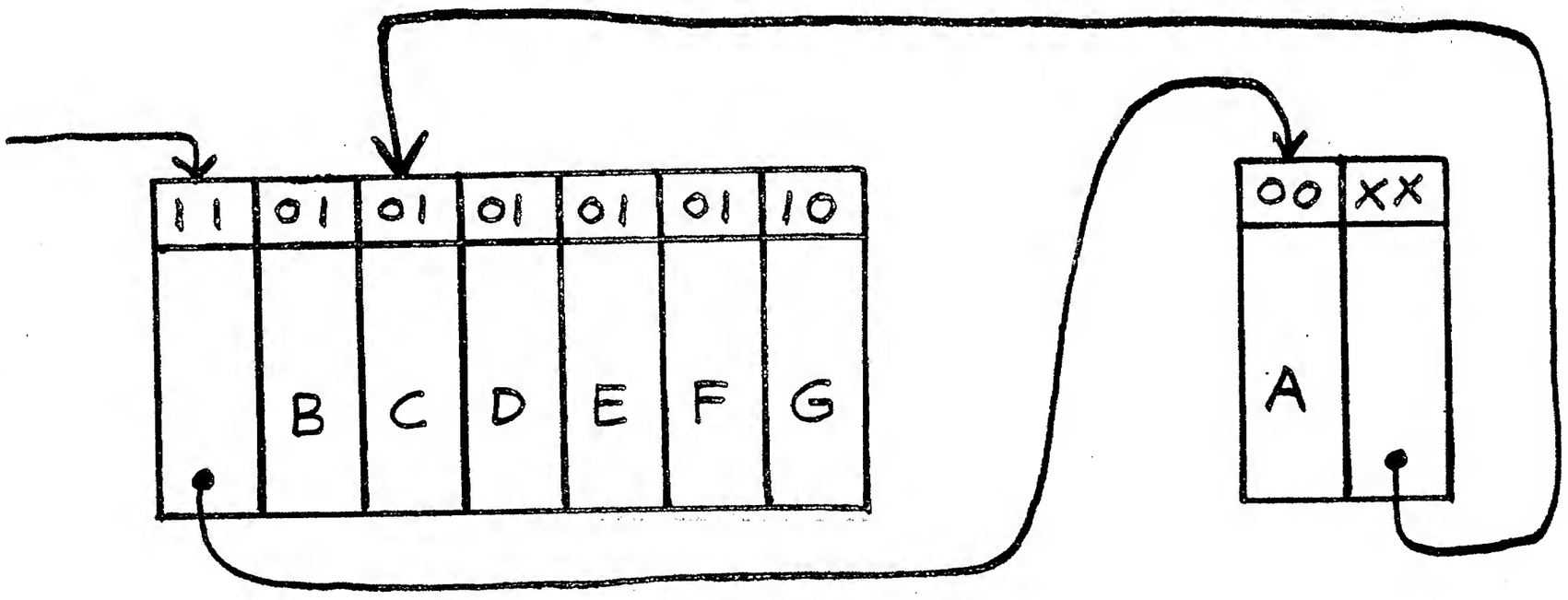

Figure 3
A Compact CDR-Coded List with One Element Spliced Out,
Using an Indirect Pointer to a New Pair

When taking the CAR of a cell one must therefore first check the CDR code; if it is 11, one must follow the indirection pointer and then take the CAR; and

similarly for the operations CDR, RPLACA, and RPLACD. The proliferation of indirect pointers is not a permanent problem, because one can organize the garbage collector (storage reclaimer) in such a way as to eliminate such pointers when reorganizing memory. The details of the CAR, CDR, RPLACA, and RPLACD operations, as well as of possible garbage collection methods, are given in [Baker].

Unfortunately, there are a few commonly-used functions which use the RPLACD operation frequently to reorder a linked list by altering CDR pointers. The best examples of these are destructive reversing (MacLISP NREVERSE, or InterLISP DREVERSE) and destructive sorting of a list. If the traditional algorithms are used straightforwardly on a CDR-coded list, then nearly all of the cells will be replaced with Indirect pointers. This causes the list to (temporarily) occupy 50% more space than in a traditional implementation, and also is wasteful of time. Moreover, the entire reason for using a destructive algorithm is to avoid copying the argument; the proliferation of Indirect pointers and the consequent copying of list cells defeats the entire purpose of using a destructive algorithm.

We shall consider here versions of the destructive reversing and sorting algorithms which take advantage of CDR-coded lists to save time and avoid creation of Indirect CDR codes. The essential idea is that in general a list will be composed of CDR-Next-coded "chunks" chained together by Normal (and possibly Indirect) pointers (see Figure 4).

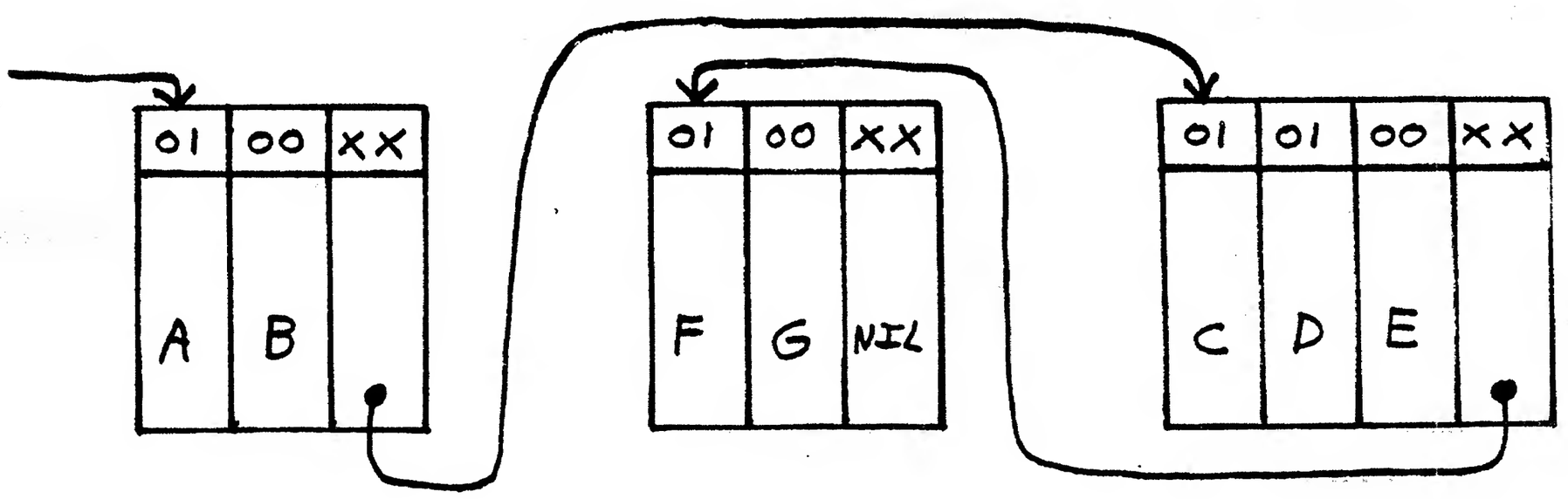

Figure 4
A Typical CDR-Coded List of Seven Elements (A B C D E F G),
Which Is Only Partially Compact

Of course, some chunks may consist of only a single CAR pointer, while others may have many contiguous CAR pointers. (Indirect pointers may also be present in the chain, but because they are invisible to CAR/CDR operations, they do not affect the general picture.)

Notice that we can consider each chunk to be a single object, a kind of array or vector, and view the entire list as a list of chunks rather than single CAR pointers. On this basis we decompose each algorithm into two levels. The higher level is isomorphic to the traditional algorithm, but operates on chunks rather than single list cells. The lower level operates on each chunk, and performs the operation by treating the chunk as an array. If all chunks are of length 1 (no Next CDR-codes occur in the list), then the lower level does nothing, and the "chunky" version of each algorithm reduces to the standard version. We shall note this in passing when appropriate.

Destructive Reversing

The traditional destructive reverse algorithm looks something like this:

```
(DEFINE NREVERSE (X)
        (DO ((A X (PROG1 (CDR A) (RPLACD A B)))
             (B NIL A))
            ((NULL A) B)))
```

Here we have used the MacLISP DO loop [Moon] {Note MacLISP DO}, which allows stepping of multiple variables. (This DO loop has a null body - the variable steppers do all the work.) This loop initializes A to X and B to NIL. It then checks the end-test (NULL A). If it succeeds (meaning A is NIL) then B is returned. Otherwise the stepping forms (PROG1 (CDR A) (RPLACD A B)) and A are evaluated, and then assigned respectively to A and B. Notice that the value of the PROG1 is (CDR A), which is evaluated before the RPLACD, and that the step value A for B is fully evaluated before the assignment to A is performed. Thus A receives its CDR, B receives A's old value, and as a side effect the old value of A has its CDR replaced by the old value of B. A roughly equivalent pseudo-ALGOL version is:

```
procedure nreverse(X);
    begin
        A := X;
        B := NIL;
        while A ≠ NIL do
            begin
                C := cdr(A);
                cdr(A) := B;
                B := A;
                A := C
            end;
        return B;
    end
```

{Note Unrolled Loop} The result of applying this algorithm to the list of Figure 1 is shown in Figure 5.

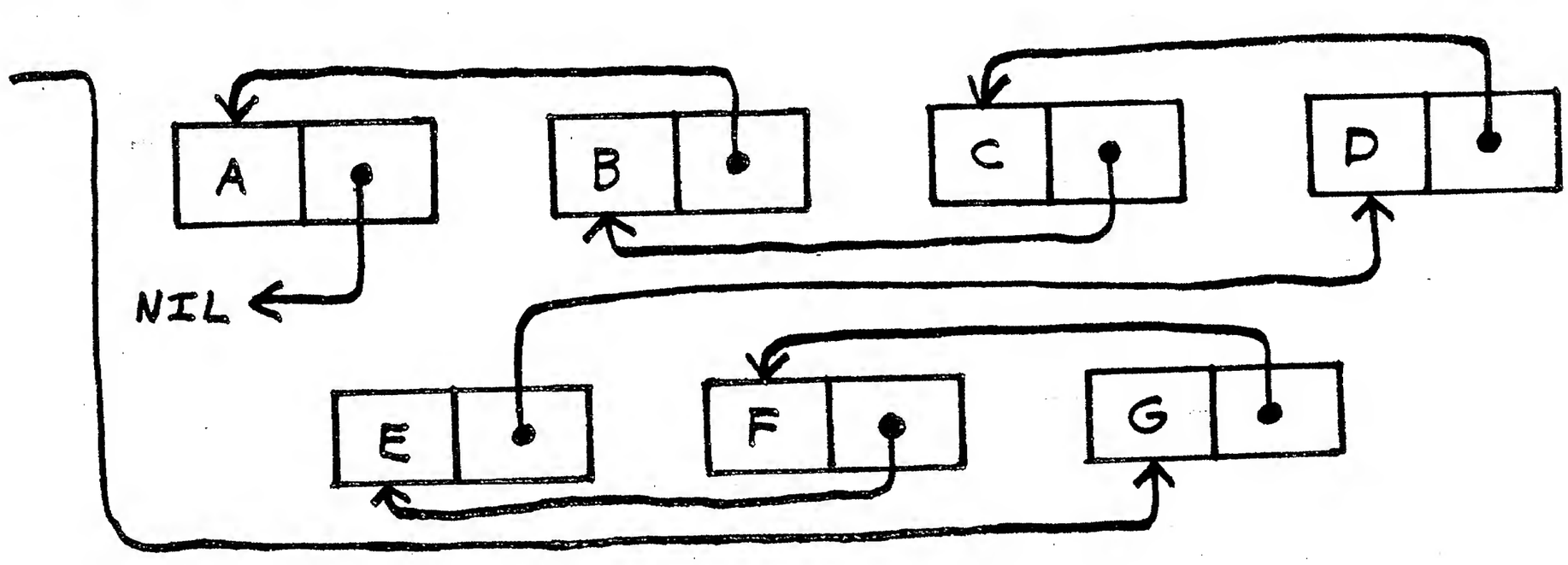

Figure 5
An Ordinary List after Destructive Reversal

If we examine our picture of a "chunky" list above, we may note that if the list of chunks is reversed, and if each chunk is reversed in place as an array, then the overall effect will be to reverse the entire list. The effect of doing this to the CDR-Coded list of Figure 4 is shown in Figure 6.

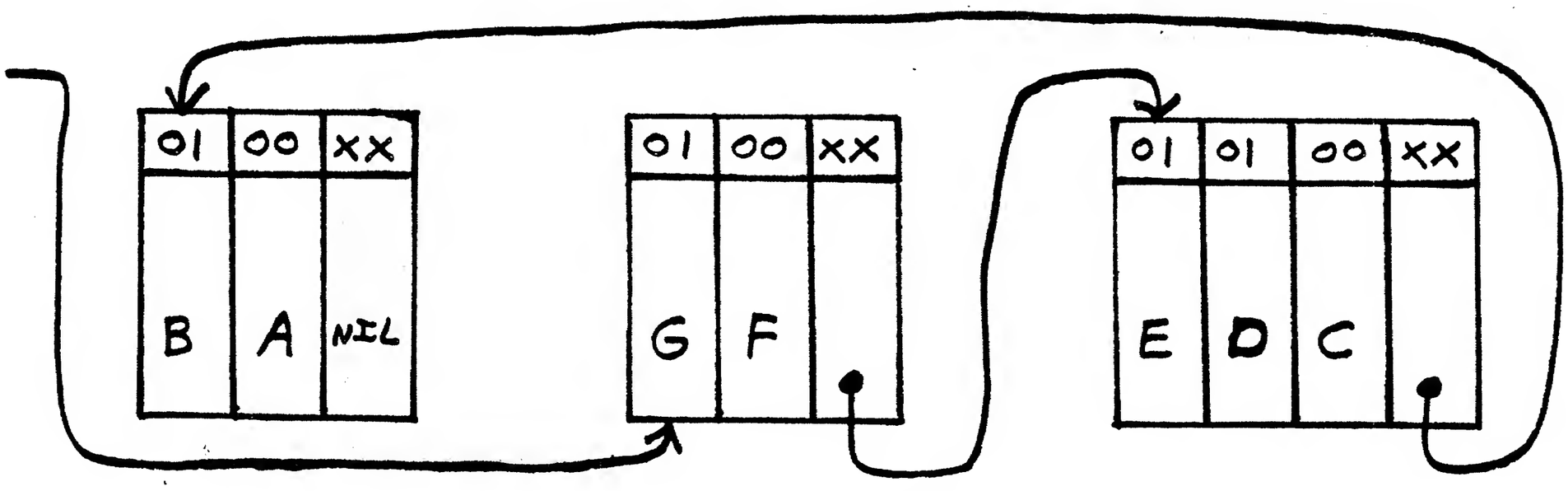

Figure 6
A Destructively Reversed CDR-Coded List, Done Cleverly

This leads us to the following algorithm:

```
procedure chunky nreverse(X);
    begin
        A := X;
        B := NIL;
        while A ≠ NIL do
            begin
                Z := A;
                comment  Find end of current chunk. ;
                while cdrcode(Z) = Next do
                    Z := cdr(Z);
                comment  Reverse this chunk as an array. ;
                for k := 0 until ⌊(adr(Z)-adr(A)-1)/2⌋ do
                    begin
                        comment  Exchange two elements. ;
                        T := car(loc(adr(A)+k));
                        car(loc(adr(A)+k)) := car(loc(adr(Z)-k));
                        car(loc(adr(Z)-k)) := T
                    end;
                comment  Now do the standard nreverse thing on the chunks. ;
                C := cdr(Z);
                cdr(Z) := B;
                B := A;
                A := C;
            end;
        return B
    end
```

The function adr takes a pointer and returns the address of the cell pointed to as a number. The function loc performs the inverse conversion, making a number into a pointer. (These are respectively called VAG and LOC in InterLISP, and MAKNUM and MUNKAM in MacLISP.) This algorithm works correctly in the face of Indirect pointers, assuming, of course, that assignments of the form car(...) := ... chase Indirect pointers properly.

If all the chunks are of length 1 (no Next codes present), then the inner while loop always terminates immediately. We assume that the for loop executes its body zero times if the specified final value is -1 ($\lfloor x \rfloor$ denotes the floor of x; $\lfloor -1/2 \rfloor = -1$, not 0). In this case we always have Z=A, and the algorithm is the same as the standard one.

The assignment "cdr(Z) := B" may cause the last cell of the last chunk to be copied if its CDR-code is Nil. Except for this no cells are copied.

This algorithm behaves a little differently from the original in that it rearranges CAR pointers as well as CDR pointers. The traditional NREVERSE has the property that a pointer into the middle of the list still has the same CAR after the operation. This property is little used in practice.

Let us analyze the running times of the two reversing algorithms. We count only the number of cells read or written; and we also assume that appropriate care is taken (by using a finite number of "machine registers") to avoid unnecessarily reading a cell twice. Thus, for example, in the chunky version the assignment "C := cdr(Z)" can take advantage of the result of the previous execution of "cdrcode(Z)".

Using the traditional algorithm on a non-CDR-coded implementation, there is one read and one write for each cell of the list.

For the traditional algorithm on a CDR-coded implementation (which will involve copying any cell whose CDR-code is Next), the cost is two reads and one write for cells which do not have CDR-Next, and one read and three writes for CDR-Next cells (all this assumes that no Indirect pointers happen to be present). Two of these writes are for the new copy of the cell. There may also be some cost associated with the garbage collection eventually required because of the consing of the copy.

The chunky version takes two reads and one write per cell, regardless of the CDR-code of the cell. Thus we see that the chunky version is no worse than the traditional version in a CDR-coded implementation, and is significantly better when a large percentage of the cells in the list to be reversed have a CDR-code of Next.

Destructive Sorting

In this application we wish to destructively sort a list by rearranging CDR pointers. (For the "chunky" version, we will allow rearranging of CAR pointers as well, as for NREVERSE.) We are given a list and an ordering predicate, and are to produce a list sorted according to the predicate. Our algorithm is based on a "traditional" list merge sort as used in MacLISP:

```
(DEFINE SORT (X PREDICATE)
        (DO ((HEIGHT -1 (+ HEIGHT 1))
             (SOFAR NIL)
             (HEADER (LIST 'HUNOZ)))
            ((NULL X) SOFAR)
            (SETQ SOFAR (MERGE SOFAR (PREFIX HEIGHT)))))

(DEFINE PREFIX (HEIGHT)
        (COND ((NULL X) NIL)
              ((< HEIGHT 1)
               (RPLACD (PROG1 X (SETQ X (CDR X))) NIL))
              (T (MERGE (PREFIX (- HEIGHT 1))
                        (PREFIX (- HEIGHT 1))))))

(DEFINE MERGE (L1 L2)
        (PROG (P)
              (SETQ P HEADER)
         A    (COND ((NULL L1) (RPLACD P L2) (RETURN (CDR HEADER)))
                    ((NULL L2) (RPLACD P L1) (RETURN (CDR HEADER)))
                    ((PREDICATE (CAR L2) (CAR L1))
                     (RPLACD P L2)
                     (SETQ P L2)
                     (SETQ L2 (CDR L2)))
                    (T
                     (RPLACD P L1)
                     (SETQ P L1)
                     (SETQ L1 (CDR L1))))
              (GO A)))
```

The SORT function repeatedly calls PREFIX, which will return a sorted list of the next 2^{HEIGHT} elements of the given list (or of all that remain if there are fewer than 2^{HEIGHT}). Notice that PREFIX uses and modifies the variable X bound in SORT. As before, PROG1 evaluates both its arguments and then returns the value of the first. SORT uses MERGE to merge the result of PREFIX with the accumulation in SOFAR, which will also be a list of length 2^{HEIGHT}. PREFIX itself uses MERGE in a similar manner. MERGE is a straightforward destructive merge of two sorted lists according to the ordering predicate. (Notice the use of HEADER as a place to begin the destructive merge. MERGE never calls itself, so the sharing of HEADER among the many calls to MERGE causes no interference. The atom HUNOZ is not relevant to the algorithm; all that matters is that we have a list cell to RPLACD. HEADER (like SOFAR) is initialized by the DO loop but is unchanged on each iteration because no stepping form is specified.) The overall effect of the SORT function is to form the elements of the given list into the leaves of a binary tree, and to form a sorted list at each interior node by merging its sons. The parameter

HEIGHT corresponds to the height of the node in the tree.

To produce a chunky SORT function, we change PREFIX to strip off chunks rather than single cells, and change MERGE to merge chained chunks. The latter is a little tricky. In the ordinary MERGE one simply compares the first two items of each list and then chooses the smaller to go into the result. The chunky MERGE cannot be this simple, because it may be that neither chunk is smaller than the other (we say that one chunk is smaller than another if and only if all its CAR elements are smaller than all CAR elements of the other; this is only a partial ordering). If neither is smaller than the other, we must sort out the elements of the two chunks so that one is lower than the other, and then proceed.

```
(DEFINE CHUNKY-SORT (X PREDICATE)
        (DO ((HEIGHT -1 (+ HEIGHT 1))
             (SOFAR NIL)
             (HEADER (LIST 'HUNOZ)))
            ((NULL X) SOFAR)
            (SETQ SOFAR (CHUNKY-MERGE SOFAR (CHUNKY-PREFIX HEIGHT)))))
```

CHUNKY-SORT is identical to SORT (except for names).

```
(DEFINE CHUNKY-PREFIX (HEIGHT)
        (COND ((NULL X) NIL)
              ((< HEIGHT 1)
               (DO ((Z X (CDR Z)))
                   ((≠ (CDR-CODE Z) Next)
                    (OR (EQ Z X)
                        (SORT-CHUNKY-ARRAY X Z))
                    (PROG1 X
                           (AND (SETQ X (CDR Z))
                                (RPLACD Z NIL))))))
              (T (CHUNKY-MERGE (CHUNKY-PREFIX (- HEIGHT 1))
                               (CHUNKY-PREFIX (- HEIGHT 1))))))
```

If HEIGHT is less than 1, then the DO loop scans down the next chunk. SORT-CHUNKY-ARRAY is assumed to sort the chunk using some algorithm appropriate to an array (such as Quicksort or Heapsort [Knuth]); however, for speed SORT-CHUNKY-ARRAY is not called if the chunk is only one element long. If all elements are only one element long, we always have Z=X, and so CHUNKY-PREFIX behaves exactly like PREFIX.

```
(DEFINE CHUNKY-MERGE (L1 L2)
        (PROG (P)
              (SETQ P HEADER)
         A    (COND ((NULL L1) (RPLACD P L2) (RETURN (CDR HEADER)))
                    ((NULL L2) (RPLACD P L1) (RETURN (CDR HEADER))))
```

```
(DO ((Z1 L1 (CDR Z1)))
    ((≠ (CDR-CODE Z1) Next)
     (DO ((Z2 L2 (CDR Z2)))
         ((≠ (CDR-CODE Z2) Next)
          (COND ((PREDICATE (CAR Z2) (CAR L1))
                 (RPLACD P L2)
                 (SETQ P Z2)
                 (SETQ L2 (CDR Z2)))
                ((OR (AND (EQ L1 Z1) (EQ L2 Z2))
                     (PREDICATE (CAR Z1) (CAR L2)))
                 (RPLACD P L1)
                 (SETQ P Z1)
                 (SETQ L1 (CDR Z1)))
                ((PREDICATE (CAR Z1) (CAR Z2))
                 (MERGE-TWO-CHUNKS-IN-PLACE
                     (ADR L1)
                     (+ (- (ADR Z1) (ADR L1)) 1)
                     (ADR L2)
                     (+ (- (ADR Z2) (ADR L2)) 1))
                 (RPLACD P L1)
                 (SETQ P Z1)
                 (SETQ L1 (CDR Z1)))
                (T
                 (MERGE-TWO-CHUNKS-IN-PLACE
                     (ADR L2)
                     (+ (- (ADR Z2) (ADR L2)) 1)
                     (ADR L1)
                     (+ (- (ADR Z1) (ADR L1)) 1))
                 (RPLACD P L2)
                 (SETQ P Z2)
                 (SETQ L2 (CDR Z2)))))))
     (GO A)))
```

If neither L1 nor L2 is NIL, then the two DO loops scan down the respective chunks, leaving Z1 and Z2 pointing to the cells containing the last CAR pointers of the chunks. Recall that at this point each chunk is completely sorted. There are then three cases to check out:
(1) The last element of chunk 2 is less than the first element of chunk 1. In this case chunk 2 is smaller than chunk 1, so chunk 2 is pulled off and placed in the result.
(2) The last element of chunk 1 is less than the first element of chunk 2. This case is symmetric to case (1). For speed, we first test whether both chunks are of length 1; if so, the fact that case (1) failed means case (2) must hold (or perhaps that the two chunks contain equal elements, which is all right), and we avoid a redundant call on PREDICATE. This test also causes CHUNKY-MERGE to be manifestly equivalent to MERGE when all the chunks are of length 1.
(3) The two chunks must be merged together in place. One chunk will receive the low elements of the merge, and one the high elements. The effect of this is to rearrange the elements so as to reduce to case (1) or case (2): the chunk receiving the low elements is then pulled off and put into the result list. Care must be taken to ensure that the chunk that receives the high elements is the one which originally had the highest element of the merged set. This guarantees that the list whose chunk receives the high elements remains totally sorted. For example, consider the two lists in Figure 7.

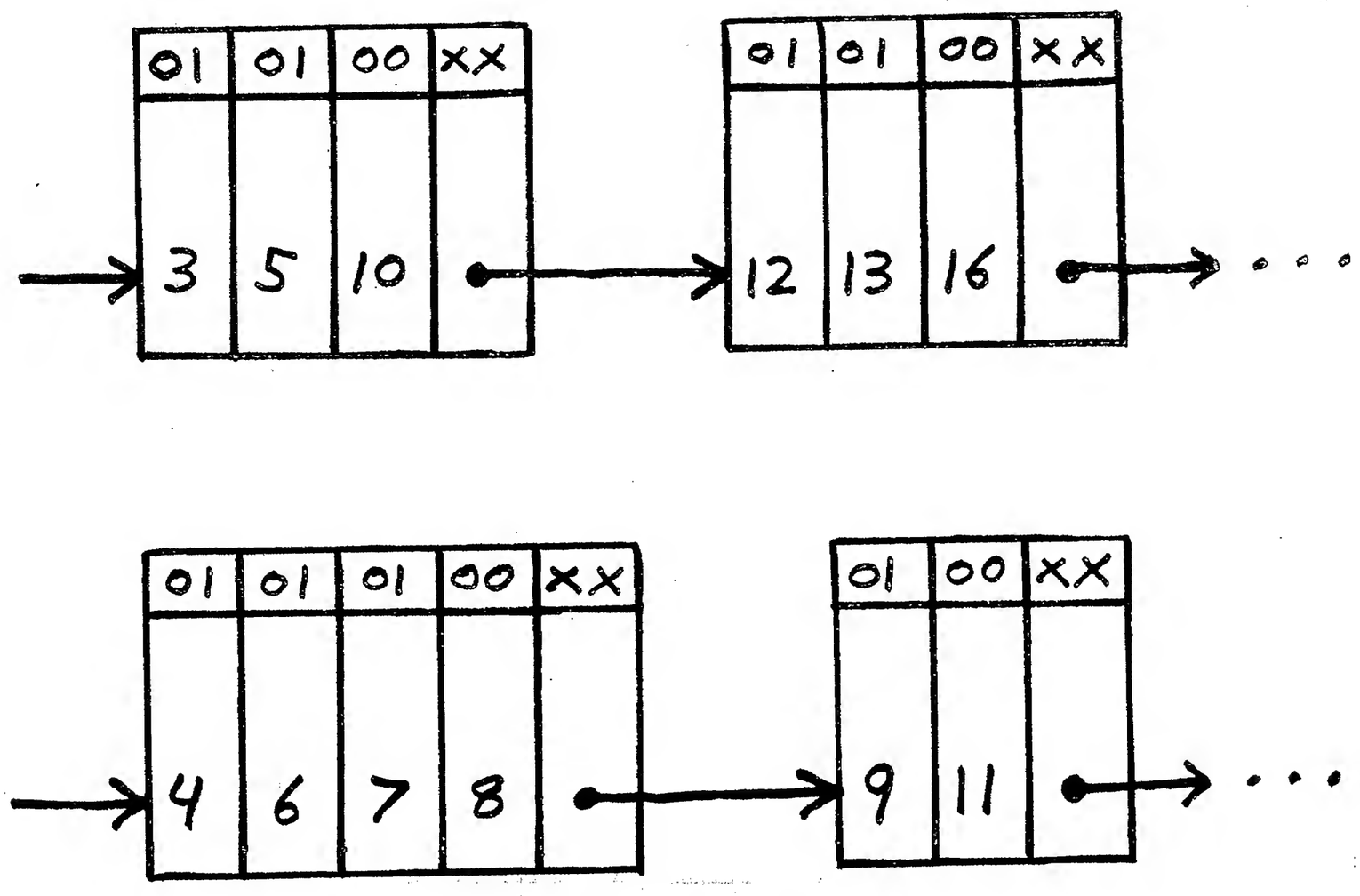

Figure 7
Two Sorted CDR-Coded Lists to be Merged

If the two leading chunks were merged with the low elements going into the first list's chunk, we would have the situation in Figure 8.

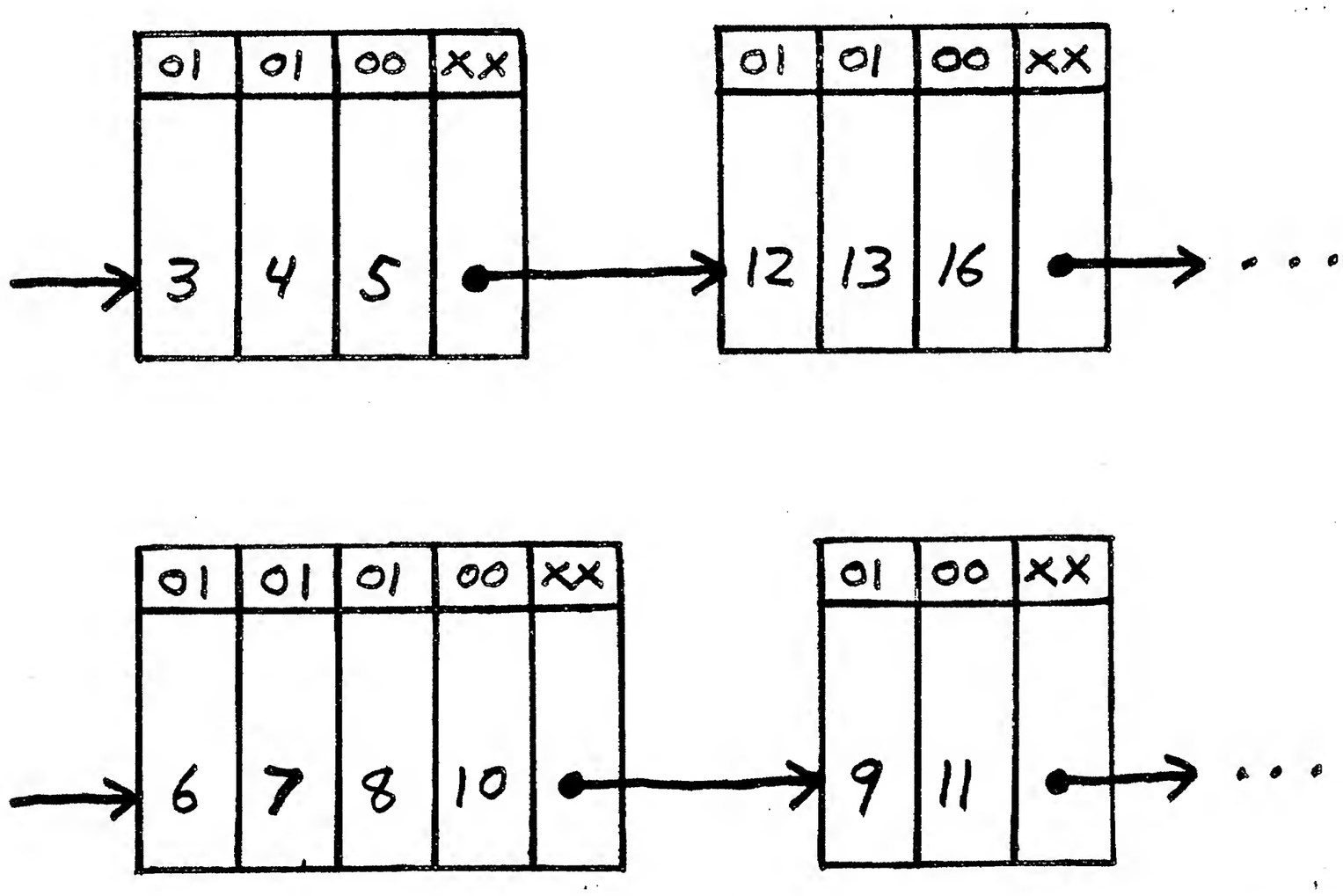

Figure 8
An Incorrect Merging of the Leading Chunks of Two Sorted Lists

This would leave the second list unsorted. After stripping off the low chunk from the first list and appending it to the output list P, we would return to the top of the merge loop with the two lists shown in Figure 9.

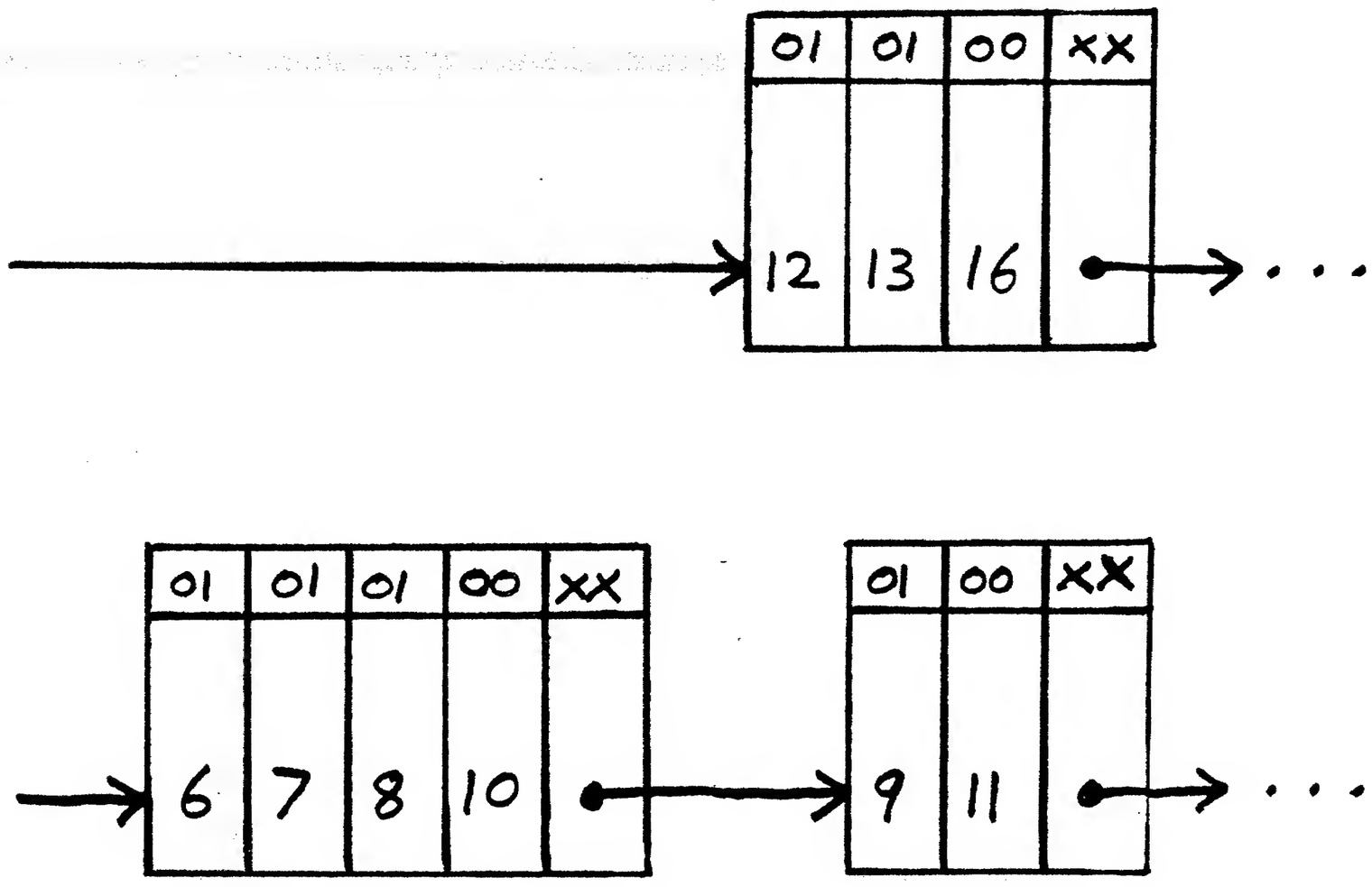

Figure 9
The Disastrous Result of an Incorrect Merge

Now the remainder of the merge could not work correctly, because the second list would not be sorted. This is the reason for the third test in the inner COND of CHUNKY-MERGE. If the chunks are chosen so that the one which originally had the largest element receives the larger elements in the merge, the lists are guaranteed to remain sorted (see Figure 10).

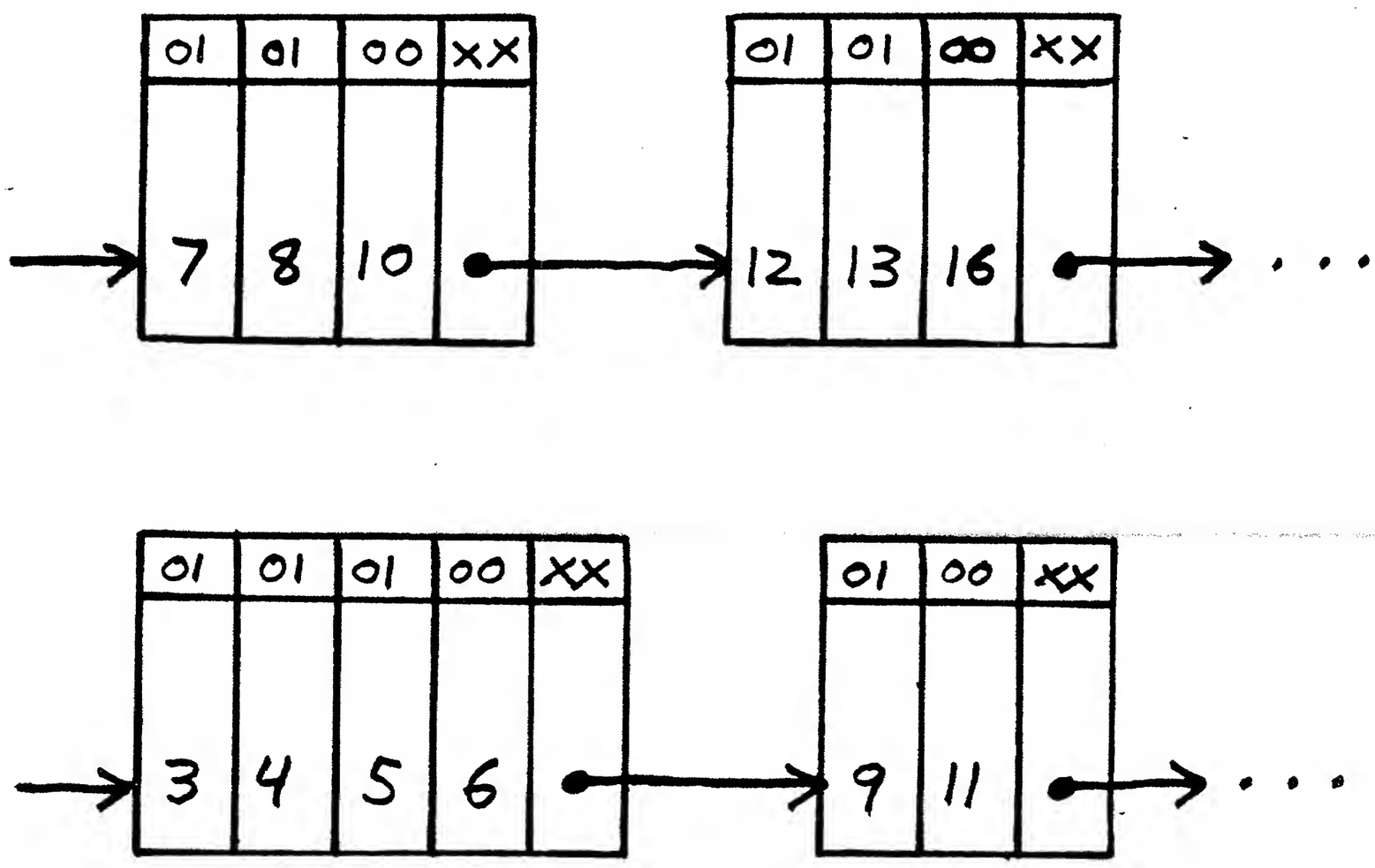

Figure 10
The Felicitous Result of a Correct Merge

```
(DEFINE MERGE-TWO-CHUNKS-IN-PLACE (M1 N1 M2 N2)
        (DO ((N (+ N1 N2 -1))
             (J N1 (+ J 1)))
            ((= J N))
            (DO ((K (- J 1) (- K 1))
                 (Q (GET-FROM-TWO-CHUNKS J)))
                ((OR (< K 0)
                     (NOT (PREDICATE Q (GET-FROM-TWO-CHUNKS K))))
                 (STORE-INTO-TWO-CHUNKS (+ K 1) Q))
                (STORE-INTO-TWO-CHUNKS (+ K 1) (GET-FROM-TWO-CHUNKS K)))))
```

This version of MERGE-TWO-CHUNKS-IN-PLACE uses a straight insertion sort [Knuth] to merge two chunks, inserting elements of the upper chunk into the lower chunk.

(There may be better ways to accomplish this. We use an insertion sort because we can take advantage of the fact that each chunk will already be sorted; hence we need only insert the elements of one into the other. Conceptually we are merging the contents of the two chunks, and storing the result back into the two chunks, one receiving all the low elements and one the high elements. However, a list merge will not do because we are trying to perform the merge in place without creating Indirect pointers.)

M1 and M2 are the addresses of the two chunks, and N1 and N2 are their lengths. J ranges from N1 (inclusive) to N (exclusive). Notice that N is N1+N2-1 rather than N1+N2. The effect of this is not to bother with the last element in doing the insertion sort. This is correct because the ordering of the chunks in case (3) of CHUNKY-MERGE guarantees that the highest element in the merge is already in place.

The functions GET-FROM-TWO-CHUNKS and STORE-INTO-TWO-CHUNKS handle the logical concatenation of the two chunks into a single array for the purposes of the insertion sort.

```
(DEFINE GET-FROM-TWO-CHUNKS (J)
        (CAR (LOC (ADDRESS-WITHIN-TWO-CHUNKS J))))

(DEFINE STORE-INTO-TWO-CHUNKS (J X)
        (RPLACA (LOC (ADDRESS-WITHIN-TWO-CHUNKS J)) X))

(DEFINE ADDRESS-WITHIN-TWO-CHUNKS (J)
        (COND ((< J N1) (+ M1 J))
              (T (+ M2 (- J N1)))))
```

The traditional sort algorithm is stable if the predicate is a "<" predicate, failing for equal keys. The chunky version fails to be stable because of the decision about chunk ordering which must be made in case (3) of CHUNKY-MERGE.

Notes

{Note MacLISP DO}
The general form of a (new-style) MacLISP DO loop is:

```
(DO ((var-1 init-1 step-1)
     (var-2 init-2 step-2)
     ...
     (var-n init-n step-n))
    (test end-1 end-2 ... end-m)
    body-1 body-2 ... body-p)
```

The initial values init-j are all evaluated, and then the variables var-j are all bound in parallel to the respective values. Then the test is evaluated. If the result is non-NIL (true), then all the end-j are evaluated, and the value of the last is returned as the value of the DO. Otherwise all the body-j are evaluated. Then all the step-j are evaluated, and all the values are assigned to the var-j; no var-j is stepped until all of the step values have been obtained. Then the test is tried again, etc. If a step-j is omitted, that variable is not stepped; it is just an initialized variable local to the DO. It is not unusual for a MacLISP DO loop to have a null body, because the stepping forms can often carry all the work.

{Note Unrolled Loop}
In practice, the traditional NREVERSE should be implemented with three copies of the loop to reduce the shuffling of variables:

```
procedure nreverse(X);  nreconc(X,NIL);

procedure nreconc(X,Y);
    begin
        A := X;
        B := Y;
        loop
            if A = NIL then return B;
            C := cdr(A);
            cdr(A) := B;
            if C = NIL then return A;
            B := cdr(C);
            cdr(C) := A;
            if B = NIL then return C;
            A := cdr(B);
            cdr(B) := C;
        repeat
    end
```

Notice that here we have defined NREVERSE in terms of NRECONC, which is a useful function in its own right:

```
(NRECONC A B) = (NCONC (NREVERSE A) B)
```

except that NRECONC is faster, avoiding having NCONC chase down the reversed list. In MacLISP on the PDP-10 this "unrolled" version of NREVERSE executes only three instructions per cell of the list (asymptotically).

Acknowledgements

An unrolled version of NREVERSE was originally coded for MacLISP by Stavros M. Macrakis. The "traditional" sort algorithm shown here was originally coded by Michael J. Fischer. A version of the CDR-coded sort algorithm was improved upon and coded for the LISP Machine by David A. Moon, and has been used for two years. Johan de Kleer provided useful comments on the content of this paper.

This work was supported in part by a National Science Foundation Fellowship, and in part by a Fannie and John Hertz Foundation Fellowship.

References

[Baker]
Baker, Henry B., Jr. List Processing in Real Time on a Serial Computer. Comm. ACM 21, 4 (April 1978), 280-294.

[LISP Machine Group]
The LISP Machine Group: Bawden, Alan; Greenblatt, Richard; Holloway, Jack; Knight, Thomas; Moon, David; and Weinreb, Daniel. LISP Machine Progress Report. AI Memo 444. MIT AI Lab (Cambridge, August 1977).

[Clark]
Clark, Douglas W., and Green, C. Cordell. "An Empirical Study of List Structure in LISP." Comm. ACM 20, 2 (February 1977), 78-87.

[Greenblatt]
Greenblatt, Richard. The LISP Machine. Artificial Intelligence Working Paper 79, MIT (Cambridge, November 1974).

[Hansen]
Hansen, Wilfred J. "Compact List Representation: Definition, Garbage Collection, and System Implementation." Comm. ACM 12, 9 (September 1969), 499-507.

[Knuth]
Knuth, Donald E. The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley (Reading, Mass., 1973).

[McCarthy]
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[Moon]
Moon, David A. MACLISP Reference Manual, Revision 0. Project MAC, MIT (Cambridge, April 1974).

[Teitelman]
Teitelman, Warren. InterLISP Reference Manual. Revised edition. Xerox Palo Alto Research Center (Palo Alto, 1975).